Compte-rendu de projet

Théorie des langages et compilation

Auteur(s) :
Romain CHARPENTIER
romain.charpentier@etu.univ-poitiers.fr

Lucille MOISE
lucille.moise@etu.univ-poitiers.fr

20 janvier 2018

0.1 Description des fonctionnalités

Pendant la réalisation de notre projet, nous avons codé un interpréteur de langage objet. Nous avons divisé le code qui est interprété en plusieurs instructions : les déclarations de variable, les déclarations de classe, les affectations et les appels de méthode. A la fin de la reconnaissance d'une instruction par l'interpréteur, celui-ci va visiter l'instruction à travers l'objet "Interpretor" qui se chargera d'exécuter l'instruction. Celui-ci va également communiquer avec une table des symboles (symbolTable) qui se chargera de sauvegarder les déclarations.

Il est ainsi possible de déclarer des variables de types primitifs et des variables objet. Les types primitifs sont boolean, float et integer. Les objets sont définis par une déclaration de classe qui est réalisée avant la déclaration dudit objet. Lors de la visite de la déclaration, l'objet Interpretor va créer des variables correspondant aux déclarations dans la table des symboles. Elles n'auront pour l'instant aucune valeur mais il est possible d'en mettre une.

```
exemple is integer; //declaration d'integer
exemple2 is exemple3; //declaration d'un objet
```

Les classes comportent 2 emplacements data et method qui vont respectivement contenir les déclarations des variables et des méthodes. Les 2 emplacements peuvent être vides. Lors de la création d'un objet, ce dernier va avoir un pointeur vers sa classe pour accéder aux méthodes et va créer des variables pour chaque déclaration de variable dans la classe. Donc chaque objet aura ses propres attributs mais partagera les mêmes méthodes avec les autres objets de même classe.

```
class exemple3 is
data
//declaration des attributs
method
//declaration des methodes
end exemple3;
```

L'emplacement method des classes contient donc des fonctions. Les fonctions peuvent être de 2 types : VoidFonction et ReturnFonction. Le premier correspond aux fonctions correspondant à une instructione et le second correspond aux fonctions renvoyant une expression (return). Les méthodes peuvent avoir également plusieurs paramètres de n'importe quel type. Et il est également possible de réaliser des affectations multiples dans les méthodes. Dans l'exemple, la method2 initialise les variables a et b à la valeur i.

```
class exemple3 is
data
        a is integer;
        b is integer;
method
        method1() is return 1;  //ReturnFonction
        method2(i : integer) is (a,b):=(i,i); //VoidFonction
end exemple3;
```

Dans le programme principal, les méthodes de cette classe pourront donc être appelées grâce à la classe Call qui contiendra toutes les informations relatives à l'appel. Ensuite l'interpréteur cherchera l'objet dans la table des symboles et à partir de celui-ci on pourra accéder à la méthode de sa classe et donc exécuter l'instruction.

0.2 Choix de structuration

Pour la structuration, les fichiers contenant la grammaire sont contenus dans le répertoire principale, tandis que notre structure se trouve donc dans le répertoire structure (les headers (fichiers .hh) sont dans le dossier headers). La grammaire va lire instruction par instruction comme détaillé précédemment. Ces instructions seront dans un Bloc qui permet de lire les instructions dans le bon sens. Les classes seront donc regroupés en plusieurs classes "majeures" Classe, Expression et Instruction. Les autres classes sont des spécificités de celles-ci et héritent donc de ces dernières. Beaucoup de vector sont utilisés dans la grammaire pour stocker les paramètres de certaines classes qui nécessitent un nombre non limité de paramètres, par exemple les paramètres d'une méthode.

0.3 Jeux d'essais

Nom du fichier	**Fonctionnalité illustrée**
fichier 1	fonctionnalité
fichier 1	fonctionnalité
fichier 1	fonctionnalité
fichier 1	fonctionnalité

TABLE 1 – Description des jeux d'essais